A Study of Atomic Action Schemes Intended for Standard Ada

A. Romanovsky
Computer Science Department
University of Newcastle upon Tyne, NE1 7RU
Newcastle upon Tyne, UK

Although the number of proposals discussing various atomic action schemes is increasing, these schemes are very rarely used in designing practical applications. To a large extent, this is accounted for by the gap existing between the languages used in research and the standard or widely spread languages (e.g. C, C++, Ada 83, Ada 95, Java) employed by practitioners. Moreover, very often researchers extend languages with new features or invent new languages to express their ideas better. Even though these approaches seem to be quite natural, they widen the gap between practice and research. To bridge this gap, we should consider fault tolerance schemes in terms of a standard language, taking the language itself for granted. The question which we believe should be addressed is how to use/implement a particular scheme in these languages rather than how to modify the language. Only in this way the schemes could be used directly and the application domains of atomic action schemes extended. The main intention of this paper is to summarise research that has been done in the last years in designing various atomic action and conversation schemes in Ada 83 and Ada 95. This should give a fuller picture of the existing schemes for researchers and help practitioners to choose the appropriate schemes. We would also like to raise and discuss some questions concerned with moving fault tolerance schemes into standard languages and environments. Finally, we intend to discuss the likely directions of future research in this area.

1. Atomic actions

Nowadays most newly designed applications are concurrent and distributed. It is widely accepted that providing fault tolerance for systems like these is a very difficult problem (Lee and Anderson, 1990). Some of the reasons are as follows: erroneous

information can easily leak between processes (nodes, objects) unless certain precautions are followed; it is far more difficult to design fault tolerance features for concurrent systems than for sequential ones; several concurrent processes should be involved in recovery, which means that they have to be designed together; processes cannot have the same view on the system state unless some special support is used. It is clear that what is required is a system design principle coupled with the corresponding run time support. To this end, the conversation scheme was introduced by B.Randell in his seminal paper (Randell, 1975). *Conversations* are essentially units of system structuring and recovery. Several processes can enter a conversation and exchange information; they establish recovery points at their entries. No information flow is allowed to cross the conversation border. The conversation participants have to leave it synchronously when the conversation acceptance test is ensured, otherwise each of them rolls back to the recovery point and they try the next alternate. Conversations can nest, in which case a set of participants from the containing conversation enter the nested one. The execution of conversations is indivisible and invisible for the outside world. The conversation concept is very general, and many different and interesting 'incarnations' intended for different applications and languages and having different properties have been designed in the last twenty years (see (Randell and Xu, 1995) for a comprehensive survey).

Further research introduced a general concept of atomic actions which can use both backward (rolling the system back to the previous correct state) and forward error recovery (with a set of exception handlers from all participants recovering the system into a correct state) (Campbell and Randell, 1986). Following (Lee and Anderson, 1990), the authors defined the atomic actions in this way: "The activity of a group of components constitutes an *atomic action* if there are no interactions between that group and the rest of the system for the duration of the activity", and we will adhere to this definition in our study. The authors proposed that the execution of fault tolerant systems be structured of atomic actions. Recovery features are associated with an action (rather than an individual process), which involves all action participants' cooperating during its recovery. With respect to forward recovery that means that if an exception has been raised in a process, then all participants of the action that this process is in should raise the same exception and call the handlers for it. If the handlers cannot do the recovering or there is no handler (even in one process) for the exception raised, then all participants signal the atomic action failure exception to the containing action. The paper discusses two approaches to dealing with the nested actions when an exception is raised in the containing one. The first is to wait until they are completed (which seems to be quite reasonable because their execution is atomic and indivisible), the second is to abort them. To do this, an additional abortion handler needs to be included in each

action participant. Another very important problem which is addressed in the paper (Campbell and Randell, 1986) is that of concurrent exception resolution for the forward recovery scheme. To deal with these exceptions, a resolution tree that imposes a partial order on all action exceptions is used. The resulting (resolving) exception is searched for in this tree as the root of the smallest subtree containing all exceptions raised.

There are two fault hypotheses, so there can be two corresponding kinds of atomic action schemes: *blocking* and *pre-emptive* ones. In blocking schemes, each participant has to either come to the action end or find an error and to inform the rest of the participants about an exception. Asynchronous schemes do not wait for this but use some feature to interrupt all participants when one of them detects an error. The appropriate approach should be chosen depending on the application, on the error which has been detected, on the failure hypothesis, etc. Recovery in blocking systems is much easier than in pre-emptive ones because each process is ready for recovery and is in a consistent state when it starts. Moreover, there is no need to program the nested action abortion for these systems because they have to be completed. Obviously, there is a danger of deadlocks stopping these systems, but we believe that careful programming with an intensive error detection would not just allow this problem to be avoided but also make the subsequent recovery simpler. There is no wasting of time in pre-emptive schemes, but the corresponding features are not readily available in many languages and systems. Even when they are, they are usually very expensive: for example, many implementations of the *asynchronous transfer of control* (discussed in details in Section 4) in Ada 95 use the two thread model with the abortion and re-creation of one thread (Burns and Wellings, 1995). Moreover, they usually have complex semantics; it is more difficult to analyse, to understand and to prove the programs that use these features. Very often some restrictions are imposed on the programming of the part that can be interrupted asynchronously as an attempt to make the implementation less expensive. For example, Ada 95 programs cannot accept messages within this part. One more difficulty with pre-emptive schemes is that the abortion of nested actions is hard to program. Some additional programming rules can improve blocking schemes and decrease time waste (time-outs; assertions; checking invariants, pre- and post-conditions; etc.). This can make possible an early detection of either an error or the abnormal behaviour of the process that has raised an exception and is waiting for the other processes.

One of the important characteristics of an atomic action scheme implementation is its centralisation. There are basically two sorts of schemes: centralised and decentralised ones. The former use an action manager which coordinates the execution of all participants. Action support for the latter schemes is represented by a set of local

supports, one in each participant. Note that the issue of scheme centralisation is not directly linked with that of distributedness: distributed schemes can be centralised and a one-machine system can use decentralised action support.

2. Atomic actions in standard languages

There appear increasingly more application domains with high dependability requirements. This situation differs a lot from that which existed when fault tolerance was launched as a independent branch of research. In those times there existed just a few exotic and very expensive fault tolerant systems. Nowadays fault tolerance is often one of the main requirements in many new applications which use standard cheap hardware and software. A lot of research has been done in fault tolerance and many elaborate schemes and algorithms have been invented over the last years. Unfortunately, they cannot be easily used in all practical systems because they require the OS and/or languages to be changed. The gap between these achievements of science and the needs of industry is getting wider. A similar trend exists in hardware: it is much cheaper and more attractive to use the standard off-the-shelf hardware (sometimes with some additional buses, comparators, synchronisers, etc.).

This is basically true for most of the existing atomic action schemes (Randell and Xu, 1995) because they are not intended for practical languages and so cannot be directly used for real applications. To solve this problem, a number of atomic action schemes have been discussed for Ada 83 (ANSI, 1983) and Ada 95 (Intermetrics, 1995) in the last years (Burns and Wellings, 1989; Clematis and Gianuzzi, 1993; Romanovsky and Strigini, 1995; Romanovsky, 1996; Wellings and Burns, 1996). We believe that this is a very important direction of research in itself as it would make the implementation of software fault tolerant systems cheaper; this 'cheapness' is a function with the use of off-the-shelf or standard components and the re-use of fault tolerance software as arguments.

All schemes that we are going to discuss are presented as sets of templates to be followed by application programmers. Sometimes these proposals describe sets of programmers' conventions guaranteeing the atomicity of actions, the absence of information smuggling and the proper use of templates. All this is obviously error prone and relies on tedious programmers' work. That is why we regard building engineering approaches to back the use of schemes like these as a very important research direction. These may include pre-processors, syntax-oriented editors,

convention checkers, macro libraries, package and procedure libraries, standard classes, using language subsets, etc. We consider a proper deep discussion of these problems to be a very important part of any scheme of this sort.

The first intention of this paper is to summarise all existing atomic action schemes which are intended for Ada 83/Ada 95, to compare them and by doing this to give an exhaustive description of the state of art in this field. Secondly, we would like to discuss future research in this direction and outline the main questions to be addressed.

The rest of the paper has the following structure. In Sections 3 and 4 we describe all atomic actions schemes found in the literature, discuss their drawbacks and advantages and outline possible improvements. In the next section we consider the colloquy scheme which relies on an Ada 83 extension and explain why we consider Ada 83 and Ada 95 to be suitable for using atomic action schemes. Section 6 discusses state restoration features which are included in some proposals. Section 7 is devoted to a thorough comparison and discussion of all mentioned schemes. We conclude by outlining several engineering approaches which make the use of these schemes more disciplined and less error prone.

Basically we assume that the readers have a general knowledge of Ada 83 and explain new features of Ada 95 briefly when necessary. At the same time we would like to give a brief description of the Ada concurrency model (ANSI, 1983). Concurrency in Ada 83 is represented by tasks which synchronise their executions and exchange information using the rendezvous model (the caller and the callee wait for each other to start the information exchange). The caller task issues an entry call and passes parameters to the callee. The callee task, which has its entry declared in the task specification, accepts the entry call by Ada statement **`accept`**. This statement has a body and when its execution is completed, the output parameters of the call are passed back to the caller, which has been blocked since it issued the entry call. Statement **`select`** can be used to construct a callee task so it can accept any one of a number of possible entry calls and/or to impose time-out on waiting for the entry to be called (using statement **`delay`**).

3. Ada 83 schemes

3.1. Atomic actions as Ada packages

The first steps, preliminary but very important, were made in the book (Burns and Wellings, 1989) in which two techniques for using atomic actions in Ada 83 were introduced. The first technique is not intended for programming any action recovery. A set of package procedures forms an action. The bodies of these procedures have a special structure: each of them starts by calling the service entry of the action controller (which is a service task that synchronises participants' exits) and calls another entry when it is about to complete the execution. All later calls are synchronised in the controller by a set of nested statements **`accept`**, that is why participants can leave the action together only when they all finish. Unfortunately, this scheme is presented as a draft without any further discussion. One very important step that is made here is combining the static and the dynamic way of structuring systems (out of packages and atomic actions, respectfully).

The second technique is intended for programming atomic actions with coordinated concurrent exception handling. This scheme also uses a service task (the action controller) with a set of nested statements **`accept`**, one for each participant. Each of them informs the controller about the code of the exception to be raised. Having received all codes, the controller raises the appropriate exception which propagates to all participants. This uses a very interesting and unique Ada 83 feature whereby an exception which is raised and not handled in the **`accept`** body is propagated to both the caller and callee. It is assumed that the execution of each participant is split into a chain of phases; at the end of each of those they have to synchronise execution and to inform the controller about the exceptions raised. So, at the end of each phase the controller executes several nested operations **`accept`**, resolves the exception in the body of the **`accept`** of the lowest level and raises a resolving exception.

Unfortunately, this approach is not exhaustive and a lot of important questions were not addressed. It is not clear when and how the service task should be started, whether service tasks should always exist for all actions that can dynamically appear and disappear, how controllers could be implemented for a task participating successively in several actions (with different participants). This scheme does not work when several actions are executed concurrently, because conventions for naming controllers and entries are not discussed properly. Periodical synchronisation is rather expensive and restrictive: it could hardly increase the scheme robustness or facilitate error detection. It does not make it possible either to detect the erroneous process or initiate its recovery. Obviously, there is no need in the additional synchronisation if the participants are executed correctly, so its use contradicts one of the main requirements that fault tolerance schemes should meet: not to decrease performance where there is no error detected. Besides, we believe that when an action can be naturally split into several

consecutive 'frames', it would be reasonable and cheap to make each of them an atomic action and to design the system as a chain of actions. This could give real benefits. The problem of predefined Ada 83 exceptions was not addressed in (Burns and Wellings, 1989), either.

The authors of this scheme rightly outlined the main problems which arise when implementing coordinated forward recovery in Ada 83. In our opinion, this is a very promising approach that makes atomic actions practical though the authors do not discuss the entire atomic action scheme or outline all peculiarities which should be taken into account to allow the scheme to be used immediately. In Section 3.4, another Ada 83 atomic action scheme will be presented that allows concurrent exception handling and resolution.

3.2. Conversations with participating tasks

A.Clematis and V.Giannuzi offer a conversation scheme (Clematis and Gianuzzi, 1993) which allows several servers and several clients to be included in a conversation and discuss the corresponding methodology for structuring concurrent programs. In the scheme, Ada 83 tasks participate in a conversation. The authors introduce the conversation manager, which is an Ada 83 task, to control the execution of the conversation. Each conversation has its manager. This manager has three entries which can be called by the tasks taking part in the conversation (to enter the conversation, to inform the manager about the result of the local acceptance test check and to receive the result of the global test check). Manager `C1M` for conversation `C1` has the following structure:

```
task C1M is                -- conversation manager
       entry REQUEST;      -- participant enters conversation C1
       entry ACC_TEST (RESULT : in BOOLEAN);
       entry TEST_LINE(RESULT : out BOOLEAN);
end C1M;
task body C1M is
       NUM_CLI: constant :=n;     -- number of clients participating in C1
       type NUM_PROC_C1 is range 0..NUM_CLI;
       NUM_ENT_PROC: NUM_PROC_C1;   -- number of clients that have entered C1
       NUM_EX_PROC: NUM_PROC_C1;    -- number of clients at the test line
       BOOL: BOOLEAN:=true;         -- overall results of local tests
begin
       NUM_EX_PROC:=0;
       NUM_ENT_PROC:=0;
       loop
              select
                     accept REQUEST do
                            NUM_ENT_PROC:=NUM_ENT_PROC+1;
                            if NUM_ENT_PROC=1 then
                                   S1.ENTER_CONV; ...; Sm.ENTER_CONV;
```

```
                        end if;
                    end REQUEST;
            or
                    accept ACC_TEST(RESULT : in BOOLEAN) do
                        NUM_EX_PROC:=NUM_EX_PROC+1;
                        BOOL:=BOOL and RESULT;
                    end ACC_TEST;
            or
                    when (NUM_EX_PROC=NUM_CLI) =>
                       accept TEST_LINE (RESULT : out BOOLEAN) do
                           NUM_ENT_PROC:=NUM_ENT_PROC-1;
                           RESULT:=BOOL;
                           if NUM_ENT_PROC=0 then
                              S1.EXIT_CONV(BOOL); ...; Sm.EXIT_CONV(BOOL);
                              NUM_EX_PROC:=0;
                              if not BOOL then
                                  NUM_ENT_PROC:=NUM_CLI;
                              end if;
                              BOOL:=true; -- ready for another attempt
                           end if;
                       end TEST_LINE;
            end select;
        end loop;
end C1M;
```

Each participating task has a fixed structure where after call REQUEST the alternates are successively tried (called) and the result of the local test is passed to the conversation manager. After the first task calls REQUEST, all servers used within the conversation (S1, ..., Sm) are involved in it until the acceptance test is satisfied (the conversation manager has to know the list of all servers used within the conversation). Each server has a fixed structure as well; it can be involved in the conversation by entry call ENTER_CONV and released by call EXIT_CONV (note that a deadlock can happen if at least two of the servers used by two conversations are the same). Each participating task calls entry ACC_TEST when it reaches the end of its alternate and checks the local acceptance test. The manager collects the results of all local acceptance tests and lets all participants leave the conversation synchronously if all tests are ensured; otherwise all of them start the next alternate.

The research that followed (Romanovsky, 1995a) proposed several improvements which may be important for different applications; what is discussed is a sort of library of templates. First of all, it is considered how a conversation can have different numbers of participants in different alternates. The second modification allows programmers to have different sets of servers in different alternates (in the original scheme all servers are kept blocked until the conversation has been completed). There is research which shows that it is not always enough to regard the global conversation test as just a conjunction of all local tests; sometimes there is a need for checking some 'non-local' conditions. That is why the original scheme was extended in this direction. Finally, the author’s (Romanovsky, 1995a) intention was to improve the robustness and control of this scheme. This could be done by complicating the conversation

manager and by imposing new conventions on programmers. All these modifications are quite natural and make the basic scheme more flexible and applicable.

The main problems of this scheme are as follows: it is not clear how to program nested conversations and how their managers should cooperate (in particular, with respect to servers and to guaranteeing that the participants of the nested actions form a subset of the participants of the containing one); the scheme (the templates) gets much more complex if some flexibility and robustness are introduced. Further research, for instance, that relying on generic packages and Ada 95 features (pointers to subprograms, object orientation, protected objects), could make the use of this scheme more comfortable and less error prone.

3.3. Conversations with tasks forking/joining

The next Ada 83 conversation scheme (Romanovsky and Strigini, 1995) uses task forking and joining. Within this scheme, the fault tolerant unit of system structuring is an Ada procedure which is built as a set of alternates each of which has the same interface as the entire fault tolerant procedure (ft-procedure). A conversation is only allowed among processes (Ada tasks) which are forked together when an alternate starts. The concurrent program is to be structured in this way and, from the outside, the ft-procedure is indistinguishable from a sequential block of execution (see Figure 1). The template of ft-procedure `Name` is as follows:

```
procedure Name(declaration of parameters) is    -- ft-procedure
-- ...
type ALTERNATE_RANGE is range 1..M;
alternate: ALTERNATE_RANGE:=1;
alternatesuccess: BOOLEAN;
-- ... temporary replicas of out parameters
begin
 loop
  case alternate is
       when 1 => Altern1( list of actual parameters, alternatesuccess);
       when 2 => Altern2( list of actual parameters, alternatesuccess);
       --  ...
       when others =>raise FAILURE;
   end case;
   exit when alternatesuccess
              and then Test(list of actual parameters);
   alternate:=alternate+1;          -- try the next alternate
 end loop;
 --  ... copy values of replicas of out parameters to out parameters
exception
 when others =>raise FAILURE;
end Name;
```

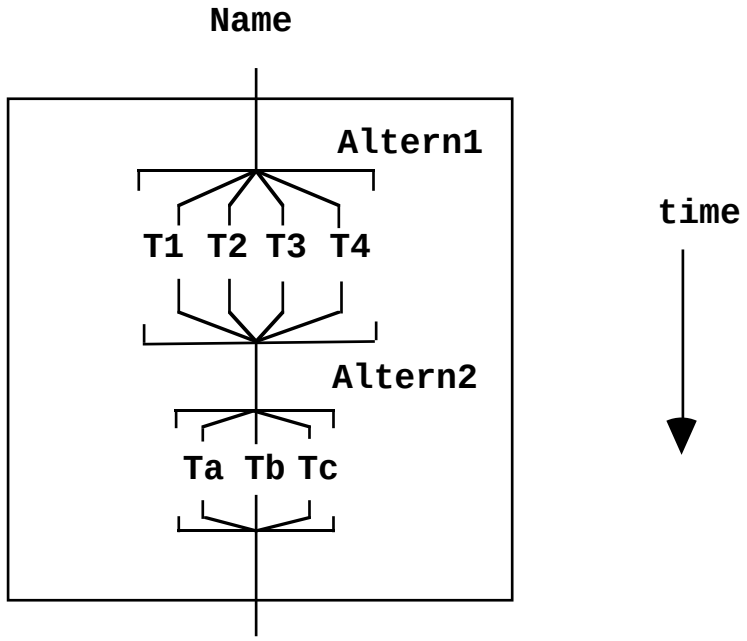

Figure 1. The execution structure of ft-procedure `Name` consisting of the sequential execution of alternate `Altern1` (with four participants forked: `T1`, `T2`, `T3`, `T4`) and, when the acceptance test failed, alternate `Altern2` (with three task forked: `Ta`, `Tb` and `Tc`).

Each alternate is to be programmed in the following way:

```
procedure Altern1( declaration of parameters of ft-procedure;
                          alternatesuccess: out BOOLEAN);

procedure Altern1( declaration of parameters of ft-procedure;
                          alternatesuccess: out BOOLEAN) is
task T11 is          -- participant of Altern1
     -- ...;
end T11;
       -- ...
task T1N is          -- participant of Altern1
       -- ...;
end T1N;
task WATCHDOG is
       entry TASK_FAILURE;
       entry TASK_DONE;
end WATCHDOG;
task body WATCHDOG is
       TaskNumber: constant:=N; -- task number in Altern1
       TimeConstr: constant:=time_out1; -- constraint for Altern1
       task_count: INTEGER:=0;
       this_alt_deadline: TIME;
begin
       this_alt_deadline:=CLOCK+TimeConstr;
       loop
             select
                   accept  TASK_FAILURE;
                   abort T11, ..., T1N;
                   alternatesuccess:=FALSE; exit;
```

```
            or    -- one task ends:
                  accept TASK_DONE;
                  task_count:=task_count+1;
                  if task_count=TaskNumber
                  then alternatesuccess:=FALSE;
                     exit; end if;
            or    -- time constraint
                  delay (this_alt_deadline-CLOCK);
                  abort T11, ..., T1N;
                  alternatesuccess:=FALSE; exit;
            end select;
      end loop;
exception
      when others => abort T11, ..., T1N;
alternatesuccess:=FALSE;
end WATCHDOG;
task body T11 is
begin
      --  ...       application code
      WATCHDOG.TASK_DONE; -- if T11 is completed without error
end T11;
--  ...
task body T1N is
begin
      --  ...       application code
      WATCHDOG.TASK_FAILURE; -- if T1N fails
      --  ...       application code
      WATCHDOG.TASK_DONE; -- if T1N is completed without error
end T1N;
begin
      null;   -- no Altern1 body
end Altern1;
```

An alternate is completed when all of its tasks have terminated. Each task can signal an error by calling entry WATCHDOG.TASK_FAILURE, in which case all tasks are aborted and the next alternate is tried. All unhandled exceptions are caught and treated as an alternate fault. Tasks and their numbers are different in different alternates. The deadline mechanism is introduced in the scheme. Time-outs can be imposed on the execution of each alternate and on the entire ft-procedure.

A service task, WATCHDOG, which plays the role of the centralised conversation (actually, alternate) manager, is started when an alternate is called. One of the reasons why it was introduced is that Ada 83, unlike Ada 95, does not allow programmers to interrupt the execution of a task asynchronously. This scheme can be much simpler in Ada 95 where a decentralised solution is possible.

One of the main concerns of the approach (Romanovsky and Strigini, 1995) is to prevent information smuggling, which is very important for all atomic action schemes as system recovery can be reduced to action recovery only if there have been no information exchanges across the action borders. Otherwise action recovery is not sufficient and the system state after it is not consistent. Information smuggling is eliminated in this scheme by restricting application programmers.

The paper (Romanovsky and Strigini, 1995) deals with the problem of using the scheme correctly and describes two ways of doing this. The first one is a set of templates, the second is an Ada 83 dialect with a pre-processor to translate the code in pure Ada 83 code and to detect the violation of any conventions. A complete set of restrictions which should be followed by programmers is included into the paper: this simple scheme works provided they are followed; otherwise it would have to be extended. The authors attempted to outline a 'safe' Ada subset which guarantees the absence of information smuggling. The scheme is suitable for only those application systems in which all concurrency is designed in the fork/joint paradigm (the software of a higher level knows nothing about the internal concurrency of the underlying software) and for those Ada 83 systems in which task creation/destruction is reasonably cheap.

3.4. Atomic actions with exception resolution

Another atomic action scheme (Romanovsky, 1996) uses the conventional Ada 83 exception mechanism and offers a set of rules and templates which make it possible to program atomic actions based on forward error recovery; Ada 83 tasks participate in atomic actions. The general approach in the paper (Campbell and Randell, 1986) is followed here. This scheme guarantees either the synchronous exit of all tasks from the actions when all of them have reached the end of exception contexts successfully, or calling handlers for the same exception (even if several tasks raise exceptions). It allows nested actions and exception propagation along nested exception contexts corresponding to the chain of nested atomic actions. Action exceptions and the corresponding values are to be declared in the following way:

```
A, B, C : exception; -- exceptions for action A0
EXC_A: constant INTEGER :=2;
EXC_B: constant INTEGER :=3;
EXC_C: constant INTEGER :=4;
```

A universal exception (Campbell and Randell, 1986) whose handler is intended for raising exception `FAILURE` should be declared and used by the participants of all actions. Besides, a special value denoting the absence of exceptions is to be declared. That is why each action uses the following declarations:

```
UNIVERSAL_EXCEPTION  : exception; -- for each action
NO_EXC: constant INTEGER :=0;
FAILURE: constant INTEGER :=1;
```

The template of the exception context for a participant of action A0:

```
begin -- start of exception context
   begin -- block for predefined exceptions
     -- ... application code
      exception
         when NUMERIC_ERROR | CONSTRAINT_ERROR |
           PROGRAM_ERROR | STORAGE_ERROR | TASKING_ERROR =>
              -- ... raising corresponding action exception
   end; -- end of additional block
exception
     when A => -- ... application code  (handler for A)
     when B => -- ... application code  (handler for B)
     when C => -- ... application code  (handler for C)
     when UNIVERSAL_EXCEPTION => -- ... application code
      -- ... raising exception of containing action
end; -- end of exception context
```

Before leaving its exception context, each task is to call a special entry to inform another task (the head process, which plays the role of the centralised manager) about its state. The state can be either normal or abnormal, the latter meaning that an exception has been raised. For example, task P2 raises exception EXC_A by calling entry RAISE_A0_P2 of task P0 (the head process):

```
P0.RAISE_A0_P2(EXC_A); -- EXC_A is passed as parameter
```

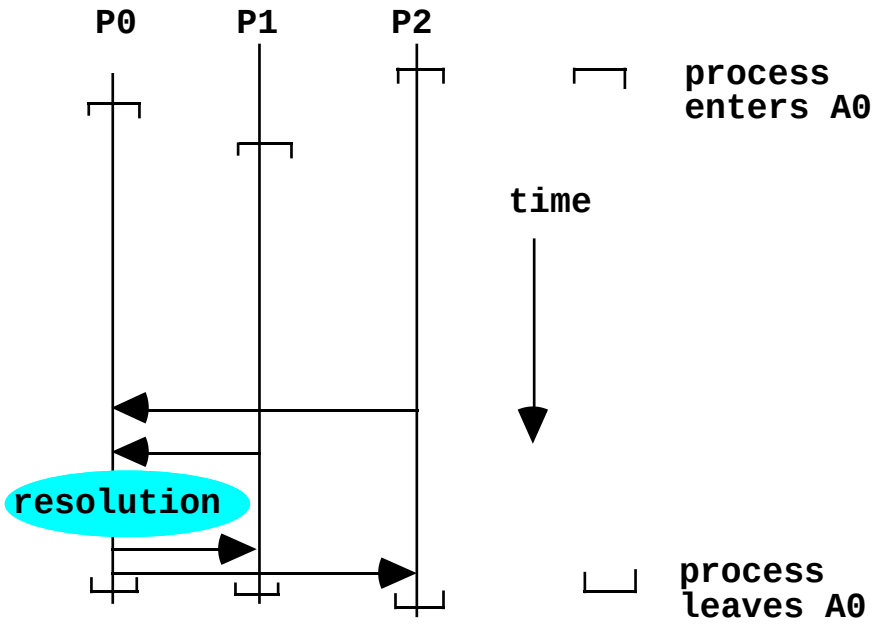

Figure 2. Action A0 with three participants P0, P1, P2 which are synchronised by the nested rendezvous at the action exit. P0 is the head process which executes the resolution procedure.

All tasks wait for each other at the exits and all exceptions raised are resolved in the head process. The resolution procedure actually raises an Ada exception which is propagated by the run time system (RTS) to all action participants through all nested rendezvous (see Figure 2). Head process `P0` in action `A0` is to have the following structure (we assume that `P0` raises `EXC_C` and that there are three participants in action `A0`):

```
accept RAISE_A0_P2(EXC_P2: in INTEGER) do
    accept RAISE_A0_P1(EXC_P1: in INTEGER) do
             -- head process calls resolution procedure:
               RESOLUTION_A0(EXC_P2, EXC_P1, EXC_C);
    end RAISE_A0_P1;
end RAISE_A0_P2;
```

Resolution procedures are programmed by the fault tolerance programmer. Within this approach, any sort of resolution procedures can be designed: they can make it possible to use simple priorities of exceptions, resolution trees or even the most general approach mentioned in the paper (Campbell and Randell, 1986), within which all action exceptions are presented as a lattice.

A set of programmers' conventions, recommendations and examples is discussed in the paper (Romanovsky, 1996). It discusses how this scheme can be moved into Ada 95 and why it can be more powerful and easier to use within this language.

The main disadvantages of the scheme are as follows: it is difficult to use when there are many processes in the action (it becomes necessary to program many nested statements **`accept`**); it would be better to use the enumeration type for atomic action exceptions rather than the integer type; the programming of resolution functions and resolution trees should be automated and made much more general, say, as an abstract data type (ADT), which would facilitate the fault tolerance programmer's job. Yet we believe that this approach is general enough to be mapped onto any existing concurrent language with 'local' exception handling (e.g. Java, Ada 95) and that it is not difficult to program a pre-processor or a convention checker which can make the use of the scheme safer.

4. Atomic actions in Ada 95

Obviously, the Ada 83 schemes discussed above are usable for Ada 95 (Intermetrics, 1995) due to Ada 83 upward compatibility. Still, some additional research is required to

adjust them to new Ada 95 features and to make them simpler and more reliable to the extent that Ada 95 allows. Now we shall discuss the paper (Wellings and Burns, 1996), which is the only one to describe schemes essentially relying on the features of Ada 95; actually, a set of schemes with different properties. There are several reasons why we believe that this study is very important. Firstly, these schemes combine the static and dynamic principles of structuring systems (out of packages and out of atomic actions, respectfully). Secondly, several of these schemes are geared for object-oriented systems and one of them is a distributed atomic action scheme programmed using the standard Ada 95 distributed features.

This paper starts with discussing the technique intended for programming simple actions. An action is encapsulated into a package with several procedures (their number is equal to that of the action participants):

```
package Simple_Action is
       procedure Actor1( parameters );
       procedure Actor2( parameters );
       --  ...
       procedure ActorN( parameters );
end Simple_Action;
```

Each procedure is called by a task, and in this way the task starts participating in the action. Ada 95 (Intermetrics, 1995) introduces a new and very important data-oriented concurrent feature: *protected objects* which are monitor-like constructs with a restricted access to their entries, procedures and functions: entries have guards (called barrier conditions), only one entry or procedure can be active at the same time, any number of functions can be active (they can have only read access to the object data). A service protected object which is defined in package body `Simple_Action` controls the action execution:

```
protected Controller is
       entry first;       -- called by Actor1
       entry second;      -- called by Actor2
       --  ...
       entry N_th;        -- called by ActorN
       entry finished;    -- called by all participants
private
       --  ...
end Controller;
```

Each procedure `ActorI` has a fixed structure in which entry `I_th` is called as the first statement of the body; the participation in the action is finished by calling entry `Controller.finished`. This simple scheme does not allow recovering and only guarantees a simultaneous exit of all participants from the action.

The second scheme, which is based on the first one, allows backward error recovery. The controller has a different structure because it has to inform all participants if one of them fails to ensure its acceptance test:

```
protected Controller is
       entry Wait_Abort;
       entry Done;
       procedure Signal_Abort;
private
       --  ...
end Controller;
```

where procedure `Signal_Abort` is called by a participant if its acceptance test is failed, entry `Done` is used when participants want to signal completion and to wait for the rest of them to finish, entry `Wait_Abort` is called to interrupt all participants if one of them signals a failing of the acceptance test (the number of participants is known to the controller). The latter feature uses the Ada 95 asynchronous transfer of control (Intermetrics, 1995), which is an extension of the Ada construct **`select`** that makes it possible to interrupt a block of operation by triggering an entry call or by time-out. Each procedure `ActorI` includes now a kind of recovery block within which several alternates are called one by one and the acceptance test is checked. The procedures have a fixed structure; their templates are thoroughly discussed in the paper. In particular, each alternate is wrapped into statement **`select`** to be interruptable asynchronously. This scheme uses a recovery cache but, unfortunately, its programming is not discussed (this problem is vital for programming all backward error recovery schemes in Ada 83/Ada 95; see Section 6 for further discussion).

The third scheme uses forward error recovery and guarantees raising the same exception in all action participants. The specification of the package declaring the action is as follows:

```
package Action is
       procedure Actor1( parameters );
       procedure Actor2( parameters );
       --  ...
       procedure ActorN( parameters );
       Atomic_Action_Failure : exception;
end Action;
```

The modified controller synchronises the exits of all participants, and if any of them has an exception raised, then the same exception will be raised in all action participants:

```
type Vote_T is ( Commit, Aborted );
protected Controller is
       entry Wait_Abort( E: out Exception_Id );
```

```
        entry Done ( Vote: Vote_T; Result : out Vote_T );
        procedure Signal_Abort( E: Exception_Id );
private
        --  ...
end Controller;
```

To do this, the exception identifier (type `Exception_Id` is predefined in Ada 95 (Intermetrics, 1995)) is passed to the controller which in its turn passes it to all participants. Asynchronous transfer of control is used here to interrupt the execution of the participants if an exception has been raised. Afterward this exception is raised in all of them and the handlers are called. The participants synchronise their exits after recovery by calling `Done`; each of them assigns value `Aborted` to parameter `Vote` to inform the controller that it was not able to handle the exception, in which case all participants raise predefined exception `Atomic_Action_Failure` in the calling tasks (which means that the action has not succeeded). It is very important that the execution of all participants is stopped by the Ada 95 RTS if one of them raises an exception (its raising is atomic); that is why there seems to be no need in exception resolution for concurrent Ada 95 systems. We would like to conclude with a warning that it may be necessary to do more research in this direction because it could be rather rough to cancel all exceptions raised concurrently (which is what the Ada 95 RTS and the scheme discussed do), so a finer approach may be more adequate.

The fourth scheme allows programming nested atomic actions; the authors describe how the third scheme can be modified to do this. A very important step is introducing atomic action type `Action_Id`:

```
package Action
        type Action_Id is private;
        function New_Action return Action_Id;
        procedure Actor1( a : Action_Id; other_parameters );
        procedure Actor2( a : Action_Id; other_parameters );
        -- ...
        procedure ActorN( a : Action_Id; other_parameters );
        Atomic_Action_Failure : exception;
private
        type Action_T;
        type Action_Id is access Action_T;
and Action;
```

This specification uses a very traditional way of introducing ADTs in Ada 83. But the implementation is very much Ada 95-oriented. The controller of each action (of a protected type) and a component of type `Action_T`:

```
type Action_T is
        record
              c : Controller_T;
               -- ...
        end  record;
```

The next scheme uses Ada 95 object orientation and allows extendability. In particular, the authors consider how the action controller of the third scheme which provides forward error recovery can be made re-usable by introducing a new tagged type (Ada 95 *tagged types* are those based on records and extendable by adding new components and by adding/overriding subprograms which are declared in this package and have one of the parameters of this type) into the private part of package `Action`:

```
package Action is
      type Action_T( At_Least : Positive ) is
                        abstract tagged limited private;
      Atomic_Action_Failure:exception;
private
      --  ...  Action_Controller
      type Action_T ( At_Least : Positive ) is tagged limited
            record
               c : Action_Controller( At_Least );
            end record;
end Action;
```

where `At_Least` is the parameter denoting the minimum number of tasks that must be active in the action for it to terminate. It is assumed here that a modification of the third scheme was made which separates the participants (procedures) and the action support. In this case, packages can be designed which will implement atomic actions for particular systems, so the code of the controller can be reused.

The last scheme which was dealt with in the paper (Wellings and Burns, 1996) is a distributed atomic action scheme. It makes extensive use of Distributed Annex (Intermetrics, 1995) which introduces the concepts of *partitions* as the units of distribution and introduces several categories of them. In particular, this action scheme uses partitions of the following categories: `Pure` (to supply types to multiple partitions), `Remote_Call_Interface` (to define the interface between active partitions and to manage global data shared by several active partitions).

The main peculiarities of distributed system programming within Ada 95 which affect the implementation of the abovementioned atomic schemes are as follows: protected objects cannot be called through partition borders; the exception identifier cannot be passed through partition borders. That is why several coordinators (controllers) are introduced into the scheme: a local controller for each participant, a distributed action controller, a shared data controller. Within this scheme, the action participants are located in different partitions `Remote_Call_Interface` (note that unlike all previous schemes, participants are not in the same package, so the package is not the action border or the unit of system structuring), and the data they share are put in a

separate partition of the same category. Package `Common_types` is a `Pure` package that provides the types which are used by all action participants. An example of a partition with shared data can be as follows:

```
with Common_Type; use Common_Type;
package Shared_Data is
       pragma Remote_Call_Interface;
       procedure Write_Data( New : in Data_Type );
       procedure Read_Data( Old : out Data_Type );
end Shared_Type;
```

The authors give a complete set of templates for programming distributed atomic actions with forward error recovery (unfortunately, the need for exception resolution is just mentioned).

One drawback of all these schemes is that participants cannot use Ada rendezvous to communicate. Other drawbacks are a vague treatment of resources as the only means of communication (we believe that the concept of atomic objects (Lynch et al., 1993) should be used here in the general case) and of state restoration features the design of which is left completely to application programmers. These schemes give wonderful examples of programming atomic actions in Ada 95 but sometimes they look bulky and the corresponding sets of conventions are difficult to follow (because of the complexity of programming atomic actions). That is why programmers would need some assistance (see Sections 7 and 8 on this).

We believe that the entire approach (Wellings and Burns, 1996) constitutes a new promising step in programming atomic actions not just in Ada 95 but in object-oriented languages on the whole. Having said that, we would like to add that more investigation is needed. It would have to solve a number of problems: state restoration, automating the use of bulky templates, introducing atomic objects, using resources and local objects within actions, exception resolution, imposing real time constraints, practical experience, etc.

5. Colloquy scheme and criticism of Ada 83 in (Gregory and Knight, 1989)

The colloquy scheme, which was introduced in the paper (Gregory and Knight, 1985), extends the original conversation scheme (Randell, 1975) in the following ways. Different processes participate in different alternates; they enter a dialog (a sort of

alternate) without knowing statically the list of the dialog participants. A set of dialogs forms a colloquy. Local acceptance tests (one for each participant) and a global acceptance test are included in this scheme; global tests are different for different dialogs. Time-outs can be imposed on the execution of each dialog and of the entire colloquy. The authors use an Ada 83 extension to demonstrate what sort of language features should be used. There are statements `discuss` which are used by tasks to declare participation in a dialog. A local test is part of this statement. A colloquy is declared as a set (Ada statement **`select`**) of attempts each of which is a `discuss`. This declaration can include time-outs.

Our general feeling is that the need in this sort of scheme does not often arise in practice. This scheme allows too much flexibility and uncertainty. It cannot be used in the standard Ada, but if applications required this sort of atomic actions, it might be worthwhile to try and map the scheme onto the standard language and to outline it as a set of corresponding templates. Still, the implementation of this scheme using Ada features seems to be cumbersome.

In their next paper (Gregory and Knight, 1989), the authors discussed problems of using conversations in production languages and, in particular, in Ada 83. The general conclusion the authors came to was that these languages are not suitable for programming conversations and that new languages should be designed. Unfortunately, it is hardly to be expected that any production language will have atomic action features. Another consideration is that we should not waste time waiting for these maybe-coming languages while there is a clear demand for programming atomic actions within the standard Ada 83/Ada 95.

Apart from this ideological disagreement, we believe that there are some misunderstandings in this paper. The first one is that there are two kinds of concurrent systems: cooperating and competing systems (Burns and Wellings, 1989; Hoare, 1976). Different paradigms are used to design these systems. Conversations are structuring units for programming cooperating systems; atomic transactions are units of which competing systems are built. Manipulating servers is obviously a competitive kind of concurrency which should be programmed using transactions. A lot of problems mentioned in (Gregory and Knight, 1989) can be easily solved within the transactional paradigm (Lynch et al., 1993): object and server sharing, action nesting, the anonymity of clients, the absence of information smuggling (see Section 3.4). The coordinated atomic (CA) action scheme (Xu et al., 1995) clearly demonstrates how atomic objects can be involved into conversations. Besides, we believe that data sharing should by no means be considered a safe way for communication and the right way for

data manipulation. We hope that using Ada 95 will allow programming atomic objects. Object creation/destruction seems to be a really important problem; we believe it can be solved by using an extended recovery cache mechanism (which apparently can be programmed in Ada 95).

6. State restoration and information smuggling

State restoration underlies all backward error recovery schemes. The states of all data need to be saved either by the underlying support or by application programmers themselves. Generally speaking, this feature can work properly only if there is no information smuggling from the components involved in the fault tolerance scheme. The exclusion of information smuggling and state restoration are complementary features. It is obvious that we want to restrict the information that has to be saved but we can do this only if it is guaranteed that there is no information exchange or 'leakage' from the components involved in recovery or the outside world. The recovery region should be made as small as possible with the help of a structuring technique, and erroneous information should not be smuggled from it. Unfortunately, there is no language which allows programmers to guarantee the absence of information smuggling. The only way is to outline a set of strict rules which should be followed by component designers. A set of these rules is discussed in (Romanovsky and Strigini, 1995). They are basically in line with good structuring techniques and disciplined design.

The general approach to the recovery cache that was discussed by F.Cristian (Cristian, 1979) can guarantee the consistency of a system and its rolling back to the previous state. It assumes that system execution is structured as a hierarchy of nested recovery regions (e.g. exception contexts or recovery blocks). The two main recovery strategies are discussed: the implicit and explicit ones. In its turn, the implicit strategy can be of two types. When the first one is used, the values of all state variables are saved into the cache before they are modified; these values can be used to recover the states if an error is detected (e.g. an exception is raised). The implicit strategy of the second type prevents the variables from being directly modified by keeping the new values in the cache. If an error is detected, the states of variables are correct, but when the end of the region is reached, the real states of all variables are to be calculated and assigned to them. Explicit recovery assumes that there are ‘cancel’ (reverse) functions for all changes made during the execution of the program region. These cancel functions are

application-dependent and have to be programmed by application programmers. The cache is used to keep pointers to them.

An Ada 83 service package `Recovery_Cache`, which is briefly discussed in (DiSanto et al., 1983), is intended for explicit recovery and includes the following subprograms:

```
Establish_recovery_point,
Discard_recovery_point,
Restore_recovery_point,
Save_prior_value,
Schedule_cancel_operation,
Prior_value_of.
```

It is clear that this package cannot be implemented by using only Ada features (the values are of different types), so it has to use the machine and RTS levels. Designing a package of this sort and general packages with the functionality discussed in (Cristian, 1979) seems to be a very important and inevitable step in programming both forward and backward error recovery schemes in Ada 83/Ada 95.

A practical approach to state restoration in Ada 83 is offered in the report (Strigini and Romanovsky, 1993). This approach is oriented towards recovery in object-based or ADT systems. An object is presented as a package with a set of subprograms and data which are stored and recovered as a whole by state restoration features. All object data are to be included into a record type as components (in the example below it is type `Spread_Data` for package `Spread_Sheet`). A service generic package, `State_Restoration`, is to be instantiated for the access type (e.g. `Spread_Pointer`); that gives the application programmer the state restoration package (`Spread_Sheet_State_Restoration`) with the service procedures to save, restore and discard the object state.

```
package Spread_Sheet is
  SprShSize : constant INTEGER:=10;
  MaxItem :    constant INTEGER:=10000;
  subtype    ItemType is INTEGER range -MaxItem..MaxItem;
  subtype    RowColumn is INTEGER range 1..SprShSize;

  type Spread_Data is private;
  type Spread_Pointer is access Spread_Data;
  -- ... subprograms to work with the spreadsheet

private
  type Type_Spread_Data is array (RowColumn, RowColumn) of ItemType;
  type Spread_Data is       -- all object data
        record
              Spread: Type_Spread_Data;
        end  record;

end Spread_Sheet;
```

Application subprograms should manipulate all object data through this access variable (of type `Spread_Pointer`) rather than directly. The responsibility of the object designer is to create (with Ada allocator **`new`**) one copy of object data by using the access type. The value of this access variable is to be assigned during the package (e.g. `Spread_Sheet`) body execution (the object initialisation phase) and to be passed to the state restoration package by calling procedure `Initialisation`. This package allows recovery regions to be nested. The implementation of stable storage used by package `State_Restoration` is separated into a special package which can use files, dual-port memory or remote nodes to keep data copies.

Another approach to state restoration which uses Ada 95 generics was mentioned in (Wellings and Burns, 1996):

```
generic
      type Data is private;
package Recovery_Cache is
      procedure Save(D : in Data);
      procedure Restore(D : out Data);
end Recovery_Cache;
```

This is only a specification of a package which has not been implemented; it has many disadvantages and is far from being usable in practice. In particular, a third operation `Discard` should be programmed and called when the action is completed (this corresponds to F.Cristian's `STANDARD_EXIT` procedure (Cristian, 1979)). A much more powerful and user-friendly support should be designed to be applicable to a lot of data at once, to save/restore/discard their states in a stable stack (to allow component nesting), etc.

7. Discussion. Object Orientation. Summary

There are two structuring units for concurrent Ada programs: tasks and packages. Clearly, fault tolerance should be expressed on one of these levels. The majority of Ada 83 schemes uses sets of either cooperating tasks or their blocks as the units of structuring cooperating systems. In either case, these sets are obviously not the first-class concepts of the language. The only exception was the scheme in (Burns and Wellings, 1989) (described first in Section 3.1), which considers that a set of procedures from the same package which are executed together is a structuring unit of cooperating concurrency. This approach seems to be very important because it is in line with designing systems of ADTs (which is a very clear concept for sequential programs

but a rather vague and not a widely accepted or used one for concurrent programs); within the approach, ADTs represent both static and dynamic units of cooperating system structuring. Although all Ada 83 schemes can be used in Ada 95 directly, we believe that further research will be required to improve them by taking into account new features of Ada 95. For example, they can be made distributed and more suitable for real time systems if the features of the Ada 95 Distributed and Real Time Annexes are used (Intermetrics, 1995). Besides, additional convention checking and programmers' job automation can be provided if Ada 95 object orientation is used.

We believe that the introduction of object orientation into Ada 95 requires new approaches to concurrent system recovery and structuring. Some first steps were made in the paper (Wellings and Burns, 1996) (they root to the only Ada 83 scheme which uses packages as the units of cooperation (Burns and Wellings, 1989) and to the general CA action scheme (Xu et al., 1995)). This allows the action controller code to be inherited (though in a restricted way). We believe that more general schemes will have to be designed that would introduce a general class `Atomic_Action` (a tagged type, maybe with abstract procedures) and allow particular actions to be created by extending it (Ada 95 offers many ways to do this). This should be in line with the general CA action concept which makes it possible to express all main features of recovery in object orientation terms. One of the problems which the approach using the general class `Atomic_Action` cannot solve is designing distributed cooperating systems because the package (all of its procedures and data) is to be located in the same location. The last scheme in (Wellings and Burns, 1996) solves this problem but action participants here are not the procedures of the same package (or the ADT). One simplistic approach could be to keep all participant procedures within one package in one location but allow other components to call them remotely by the remote procedure call (which the Ada 95 Distributed Annex allows).

Another object-oriented approach which could be attempted in the future research of Ada 95 is the scheme in the paper (Romanovsky, 1995b), which regards a set of procedures from different packages (ADTs) as the unit of cooperation. This can essentially simplify the implementation of distributed atomic actions in Ada 95 but contradicts the idea of designing these actions as tagged types.

Of all papers discussed here only two (Romanovsky and Strigini, 1995; Romanovsky, 1996) give a complete set of programmers' conventions and address the engineering problems of the scheme use. We consider this to be the main disadvantage of this research direction as a whole, because it is obviously not enough to show how an

atomic action scheme can be programmed; the purpose is to help programmers to avoid all problems caused by the necessity of following a set of conventions.

Now we would like to summarise the main peculiarities of Ada 83 schemes briefly. These schemes have many characteristics in common: a static number of participants, a centralised manager. They are neither object-oriented nor distributed; they cannot be extended. Table 1 compares them.

Table 1. Ada 83 schemes

	forward/ backward error recovery	pre-emptive/ blocking scheme	detailed/ general	synchronous/ asynchronous entry	time-out facility	structuring units	nesting
atomic action packages (Burns and Wellings, 1989)	no	blocking	general description	asynchronous	no	package	no
atomic action packages with forward recovery (Burns and Wellings, 1989)	forward	blocking	general description	asynchronous (intermediate synchroni-sations are required)	no	package	no
conversations with participating tasks (Clematis and Gianuzzi, 1993)	backward, recovery points are programmed by application programmers	blocking	detailed proposal	asynchronous	no	set of tasks	no
conversations with tasks forking/ joining (Romanovsky and Strigini, 1995)	backward, no needs in recovery points	pre-emptive	detailed proposal	synchronous	yes	procedure	yes
atomic actions with exception resolution (Romanovsky, 1996)	forward with exception resolution	blocking	detailed proposal	asynchronous	no	set of tasks	yes

Let us discuss the main peculiarities of Ada 95 schemes. None of these schemes has a time-out facility. All of them use asynchronous entry and use the package as the unit of concurrent system structuring, so tasks call procedures and, by doing this, enter the action. In all of these schemes, rendezvous cannot be used for these participants to communicate, so some shared data/resources, objects should be used (this is not discussed in the paper). All of them use a centralised manager. Only some basic points

are discussed for all of these schemes in the paper (Wellings and Burns, 1996) but many essential questions are left to programmers. In particular, it is not clear how programmers' job can be made simpler and more reliable. Table 2 summarises these Ada 95 schemes.

Table 2. Ada 95 schemes (Wellings and Burns, 1996)

	forward/ backward error recovery	pre-emptive/ blocking scheme	extendibility	re-usability	action unit	nesting
simple atomic actions	no	blocking	not extendable	not reusable	package	no
atomic actions with backward recovery	backward, recovery points are programmed by application programmers	pre-emptive	not extendable	not reusable	package	no
atomic actions with forward recovery	forward, no exception resolution	pre-emptive	not extendable	not reusable	package	no
atomic actions with forward recovery and action nesting	forward, no exception resolution	pre-emptive	not extendable	reusable (action controller is type)	instance of a type	yes
object-oriented atomic actions with forward recovery	forward, no exception resolution	pre-emptive	extendable	reusable (action controller is type)	instance of a tagged type	no
distributed atomic actions	forward, no exception resolution	pre-emptive	not extendable	reusable (action controller is type)	set of partitions with one task in each	no

It is important to bear in mind that the appropriate atomic action scheme is to be chosen by the fault tolerance programmer depending on the peculiarities of the application. For example, the application most suitable for real time seems to be the scheme in the paper (Romanovsky and Strigini, 1995) albeit task creation may be rather expensive; the only forward error recovery schemes are in (Romanovsky, 1996) in Ada 83 and (Wellings and Burns, 1996) in Ada 95; the scheme in (Clematis and Gianuzzi, 1993) allows servers to be used within conversations; the only backward error recovery scheme which does not assume that the programmer is to implement the recovery cache and take care of state restoration is (Romanovsky and Strigini, 1995), etc.

8. Safeguarding use of atomic actions

As mentioned in Section 2, all these atomic action schemes are presented as sets of templates which should be followed by application programmers. Besides, some of the proposals describe sets of programmers' conventions which guarantee the atomicity of actions, the absence of information smuggling and the proper use of templates. This approach is obviously error prone and tedious. That is why we regard building engineering approaches for backing the use of schemes like these as a very important research direction. These may include pre-compilers, syntactic editors, convention checkers, macro libraries, package and procedure libraries, standard classes, using language subsets, etc.

Pre-compiling allows new language constructs to be introduced into Ada 83 and Ada 95. For instance, we outlined a conversation scheme above which allows declaring variants, acceptance tests, failure actions by new language constructs (Romanovsky and Strigini, 1995). A deep discussion of the disadvantages of this approach can be found in (Randell, 1993); the main one is that programmers have to use two levels because the standard Ada (debuggers, linkers, compilers, library support) does not operate on the extended language level. That is why we do not consider this approach the most suitable.

More traditional approaches can rely on using package and procedure libraries. Although none of the abovementioned schemes allows this, we believe that the right direction of research is to implement a wide range of different conversation and atomic action schemes of which programmers can take their choice (maybe with some adjustments) considering the peculiarities of a particular application.

A very important and promising way which is suitable for Ada 95 relies on using inheritance by creating atomic action class libraries (see the discussion in Section 8). Generally speaking, all schemes mentioned above can be thought of as implementations within class libraries. This approach will make it possible to re-use atomic action schemes within the standard Ada 95 and to help programmers to follow conventions and to avoid mistakes. This is due to the fact that the essential part of the scheme structure can be hidden in the service class. Another reason is that programmers have to think only about some parts of the scheme (e.g. alternates, variants, acceptance tests) which can be the abstract (the Ada 95 analogue for C++ ‘virtual’) subprograms of the base service class, and the strong typing allows a strict check during the compiler time.

Programmers' conventions are a very popular way to assist programmers. Unfortunately, it is error prone. Generally speaking, a sort of guidebook should be offered to application programmers which would discuss a set of conventions for them

to follow. Templates (skeletons), restrictions aimed at avoiding information smuggling and guaranteeing the atomicity property, examples, warnings, discussions of important details should show how these schemes could be programmed and used in a better way. This guide is to be prepared by system or fault tolerance programmers; it would explain how and when to use atomic action schemes and would be regarded as a 'law' for teams of application programmers. One feature that could automate the use of this guide is an extended syntactic editor. For example, very often text editors allow language-oriented editing/checking by macro-processing to be introduced, and this could be used for inserting some templates into the source code.

Convention checkers could be used together with programmers' conventions to check whether they can be breached. So, this guide could work together with a program that would check that everything is in accordance with the guide book. Another complementary feature for programmers' conventions is run time checkers - an 'extended' language RTS: additional tasks to check invariants, controlling information stream, monitoring applications, checking data consistency, etc. Ada 83/Ada 95 offers several linguistic mechanisms which can be used here: tasks, package libraries, asynchronous transfer of control, etc. (see Section 2).

A well-balanced combination of the above mentioned approaches (which are to a great extent complementary) should be chosen for a particular application. All these approaches have several important advantages: they are cheaper than pre-processing or changing language/compilers; they are more convenient for programmers; there is no need to modify the standard software and to implement new engineering features (e.g. debuggers, linkers, or, which is very important for critical applications, validation/verification tools); all software remains re-usable, etc.

It is worth noting that many of these approaches (language subsetting, pre-processing, additional checking tools, manual transformation into implemented languages) were recognised as important ways of improving the properties of existing languages without modifying them (Horning, 1979). We believe that the experience and the results which have been gained here can simplify the introduction of atomic action schemes into Ada.

9. Conclusions

The main intention of this paper is to summarise the research which has been done over the last years in designing atomic action and conversation schemes in Ada 83 and Ada

95. We believe that it gives a comprehensive picture of the state of art in this field which should help practitioners to choose appropriate schemes and researchers to design schemes with new properties. Apart from this, we have raised and discussed the questions of moving fault tolerance schemes into the standard Ada 83/Ada 95 languages and environments, which, as we have tried to prove, is of great importance for the applicability of the entire software fault tolerance. Our final intention was to discuss likely directions of future research in this area.

We deliberately did not survey the proposals discussing any Ada extension but included only the two of them that seem to be important for our purposes and allow a better understanding of the matter.

We foresee a considerable amount of research proposing new atomic action schemes in Ada 95, and we believe that our paper makes a twofold contribution to this end. It offers a complete discussion of all existing schemes. Besides, it emphasises the need in discussing software engineering approaches which should make the use of these schemes simpler, less error prone and more disciplined, and offers some of those.

Acknowledgements. This research was supported by the DeVa (Design for Validation) Basic ESPRIT Project and by the Royal Society Joint Project No.638072. Thanks must go to my colleagues involved in these projects: B.Randell, J.Xu, R.Stroud, A.Zorzo and to the anonymous referee.

References:

ANSI, ANSI/MIL-STD-1815a Reference Manual for the Ada Programming Language, American National Standards Institute, 1983.

Burns, A. and Wellings, A., *Concurrency in Ada*, Cambridge University Press, 1995, p. 396.

Burns, A. and Wellings, A. J., *Real-time Systems and their Programming Languages*, Addison-Westley, New York, 1989, p. 575.

Campbell, R. H. and Randell, B., Error Recovery in Asynchronous Systems, *IEEE Trans. on Software Eng.* SE-12, 8, 811-826 (1986).

Clematis, A. and Gianuzzi, V., Structuring conversation in operation/procedure oriented programming languages, *Computer Languages* 18, 3, 153-168 (1993).

Cristian, F., A Recovery Mechanism for Modular Software, in *Proceedings of the 4th International Conference on Software Engineering*, Munich, 1979.

DiSanto, M., Nigro, L. and Russo, W., Programming Reliable and Robust Software in Ada, in *Proceedings of the 13th International Fault Tolerant Computing System Symposium*, Italy, 1983.

Gregory, S. T. and Knight, J. C., A New Linguistic Approach to Backward Error Recovery, in *Proceedings of the 15th International Symposium on Fault-Tolerant Computing*, Michigan, 1985.

Gregory, S. T. and Knight, J. C., On the Provision of Backward Error Recovery in Production Programming Languages, in *Proceedings of the 19th International Symposium on Fault-Tolerant Computing*, Chicago, Illinois, 1989.

Hoare, C. A. R., Parallel Programming: an Axiomatic Approach, in *Languages Hierarchies and Interfaces, Lecture Notes in Computer Science, LNCS-46*, (G. Goos and J. Hartmaur, eds.), Springer-Verlag, 1976.

Horning, J. J., Programming Languages, in *Computing Systems Reliability*, (T. Anderson and B. Randell, eds.), Cambridge University Press, 1979.

Intermetrics, Information technology - Programming languages - Ada. Language and Standard Libraries. ISO/IEC 8652:1995(E), Intermetrics, Inc., 1995.

Lee, P. A. and Anderson, T., *Fault Tolerance: Principles and Practice*, Springer-Verlag, Wien - New York, 1990, p. 320.

Lynch, N. A., Merrit, M., Weihl, W. E. and Fecete, A., *Atomic Transactions*, Morgan Kaufman, San Mateo, 1993, p. 500.

Randell, B., System Structure for Software Fault Tolerance, *IEEE Trans. on Software Eng.* SE-1, 2, 220-232 (1975).

Randell, B., Approaches to Software Fault Tolerance, in *Proceedings of the 25th Annual LAAS Conference*, Toulouse, France, 1993.

Randell, B. and Xu, J., The Evolution of the Recovery Blocks, in *Software Fault Tolerance*, (M. R. Lyu, ed.), John Wiley & Sons, 1995.

Romanovsky, A., Application specific conversation schemes for ADA programs, *Microprocessing and Microprogramming* 41, 10, 703-713 (1995a).

Romanovsky, A., Conversations of Objects, *Computer Languages* 21, 3/4, 147-163 (1995b).

Romanovsky, A. and Strigini, L., Backward error recovery via conversations in Ada, *Software Engineering Journal* 10, 8, 219-232 (1995).

Romanovsky, A., Practical exception handling and resolution in concurrent programs, Technical Report 545, Computing Department, University of Newcastle upon Tyne, 1996.

Strigini, L. and Romanovsky, A., Implementing atomic transactions in Ada, Technical Report (unpublished) IEI CNR, Pisa, Italy, June, 1993.

Wellings, A. J. and Burns, A., Implementing Atomic Actions in Ada95, Technical Report YCS-263, Department of Computer Science, University of York, 1996.

Xu, J., Randell, B., Romanovsky, A., Rubira, C., Stroud, R. and Wu, Z., Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery, in *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, California, 1995.